What's new in Airflow 2

Apache Airflow Online Summit
8th of July 2020
Who are we?

Tomek Urbaszek
Committer, PMC Member
Software Engineer @ Polidea

Jarek Potiuk
Committer, PMC member
Principal Software Engineer @ Polidea

Kamil Breguła
Committer, PMC member
Software Engineer @ Polidea

Ash Berlin-Taylor
Committer, PMC member
Airflow Engineering Lead @ Astronomer

Daniel Imberman
Committer, PMC Member
Senior Data Engineer @ Astronomer

Kaxil Naik
Committer, PMC member
Senior Data Engineer @ Astronomer
Announcements

New PMC members

Tomek Urbaszek
Committer, **PMC Member**
Software Engineer @ Polidea

Daniel Imberman
Committer, **PMC Member**
Senior Data Engineer @ Astronomer

Kamil Breguła
Committer, **PMC member**
Software Engineer @ Polidea

New committer

QP Hou
Committer
Senior Engineer @ Scribd

*Talk: Teaching an old DAG new tricks*
Friday July 10 th,  5 am UTC
“Ask Me Anything” session with Airflow PMCs

- Asia friendly time-zone
- Thursday 11 pm PDT / Friday 6 am UTC
- Hosted by Bangalore Meetup
- BYO questions
High Availability
Scheduler High Availability

Goals:

- Performance - reduce task-to-task schedule "lag"
- Scalability - increase task throughput by horizontal scaling
- Resiliency - kill a scheduler and have tasks continue to be scheduled
Scheduler High Availability: Design

- Active-active model. Each scheduler does everything
- Uses existing database - no new components needed, no extra operational burden
- Plan to use row-level-locks in the DB (SELECT ... FOR UPDATE)
- Will re-evaluate if performance/stress testing show the need
Example HA configuration
Scheduler High Availability: Tasks

- Separate DAG parsing from DAG scheduling ✔
  
  This removes the tie between parsing and scheduling that is still present

- Run a mini scheduler *in the worker* after each task is completed ✔
  
  A.K.A. "fast follow". Look at immediate down stream tasks of what just finished and see what we can schedule

- Test it to destruction - In progress
  
  This is a big architectural change, we need to be sure it works well.
Measuring Performance

Key performance we define as "Scheduler lag":

- Amount of "wasted" time not running tasks
- \( \text{ti.state_date} - \max(\text{t.end_date for t in upstream_tis}) \)
- Zero is the goal (we'll never get to 0.)
- Tasks are "echo true" -- tiny but still executing
**Preliminary performance results**

Case: 100 DAG files | 1 DAG per file | 10 Tasks per DAG | 1 run per DAG

Workers: 4 | Parallelism: 64

**1.10.10:** 54.17s (σ19.38) Total runtime: 22m22s

**HA branch - 1 scheduler:** 4.39s (σ1.40) 1m10s

**HA branch - 3 schedulers:** 1.96s (σ0.51) Total runtime: 48s
Preliminary performance results

Case: 1 Dag File | 1 Dag Per File | 20 Tasks per DAG | 1000 runs per DAG

Workers: 30 | Parallelism: 40960 | Default pool size 40960

1.10.10: 42.14s (σ7.06) Total runtime: 1h 30m 14s

HA branch - 1 scheduler: 0.68s (σ0.19) Total runtime: 18m 51s

HA branch - 3 schedulers*: 1.54s (σ1.79) Total runtime: 12m 52s
DAG Serialization
Dag Serialization

(1) Vanilla Airflow

(2) Airflow with DAG Serialization
Dag Serialization (Tasks Completed)

- **Stateless Webserver:** Scheduler parses the DAG files, serializes them in JSON format & saves them in the Metadata DB.

- **Lazy Loading of DAGs:** Instead of loading an entire DagBag when the Webserver starts we only load each DAG on demand. This helps *reduce Webserver startup time and memory*. This reduction in time is notable with large number of DAGs.

- Deploying new DAGs to Airflow - no longer requires long restarts of webserver (if DAGs are baked in Docker image)

- Feature to use the “JSON” library of choice for Serialization (default is inbuilt ‘json’ library)

- Paves way for **DAG Versioning & Scheduler HA**
Dag Serialization (Tasks In-Progress for Airflow 2.0)

- Decouple DAG Parsing and Serializing from the scheduling loop.
- Scheduler will fetch DAGs from DB
- DAG will be parsed, serialized and saved to DB by a separate component “Serializer”/ “Dag Parser”
- This should reduce the delay in Scheduling tasks when the number of DAGs are large
DAG Versioning
Dag Versioning

Current Problem:

- Change in DAG structure affects viewing previous DagRuns too
- Not possible to view the code associated with a specific DagRun
- Checking logs of a deleted task in the UI is not straight-forward
Dag Versioning (Current Problem)

```python
from airflow.models.dag import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

with DAG('example_dag_1_1', schedule_interval=None,
         start_date=datetime(2020, 4, 25)) as example_dag_1_1:

    task_1 = BashOperator(
        task_id='task_1',
        bash_command='echo hello',
    )

    task_2 = BashOperator(
        task_id='task_2',
        bash_command='echo hello',
    )

task_1 >> task_2
```
Dag Versioning (Current Problem)

```python
from airflow.models.dag import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

with DAG('example_dag_1_1', schedule_interval=None,
         start_date=datetime(2020, 4, 25)) as example_dag_1_1:

    task_1 = BashOperator(
        task_id='task_1',
        bash_command='echo hello',
    )

    new_task = BashOperator(
        task_id='new_task',
        bash_command='echo hello',
    )

    task_2 = BashOperator(
        task_id='task_2',
        bash_command='echo hello',
    )

    task_1 >> new_task >> task_2
```

New task is shown in Graph View for older DAG Runs too with “no status”.

Airflow DAGs with new task added.
Dag Versioning

Current Problem:

- Change in DAG structure affects viewing previous DagRuns too
- Not possible to view the code associated with a specific DagRun
- Checking logs of a deleted task in the UI is not straight-forward

Goal:

- Support for storing multiple versions of Serialized DAGs
- Baked-In Maintenance DAGs to cleanup old DagRuns & associated Serialized DAGs
- Graph View shows the DAG associated with that DagRun
Performance Improvements
Components performance improvements

- Focus on the current code
  - Reviews each component in turn
- Tools supporting performance tests - *perf_kit*

```python
@timing()
def test_dag_sync():
    with count_queries():
        DAG.bulk_sync_to_db()
```
Avoid loading DAGs in the main scheduler loop
Limit queries count

**DagFileProcessor:**
When we have one DAG file with 200 DAGs, each DAG with 5 tasks:

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time:</td>
<td>8080.246 ms</td>
<td>628.801 ms</td>
<td>-7452 ms (92%)</td>
</tr>
<tr>
<td>Queries count:</td>
<td>2692</td>
<td>5</td>
<td>-2687 (99%)</td>
</tr>
</tbody>
</table>

**Celery Executor:**
When we have one DAG file with 200 DAGs, each DAG with 5 tasks:

<table>
<thead>
<tr>
<th></th>
<th>Postgres Before</th>
<th>Postgres After</th>
<th>Redis Before</th>
<th>Redis After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time</td>
<td>3.1 s</td>
<td>27.825 ms</td>
<td>778.557 ms</td>
<td>3.417 ms</td>
</tr>
<tr>
<td>Queries count</td>
<td>5000</td>
<td>1</td>
<td>5000</td>
<td>1</td>
</tr>
</tbody>
</table>
How to avoid regression?

```python
with assert_queries_count(3):
    DAG.bulk_sync_to_db(dags)
```
API: follows Open API 3.0 specification

Outreachy interns

Ephraim Anierobi

Omair Khan
## API development progress

### API Endpoints
- **Read - Connection**: #8127 - Done
- **Read - DAG Model**: #8128 - Community review
- **Read - DAG Runs**: #8129 - Done
- **Read - Task Instance**: #8132 - Development in progress
- **Read - Variable**: #8133 - Done
- **Read - XCOM**: #8134 - Done
- **Dag source**: #8137 - Community review
- **Dags structure/Task**: #8138 - Done

### Community tasks
- **High level info**: #8107 - Done
- **Select OpenAPI spec**: #8108 - Done
- **Basic integration airflow and connection**: #8109 - Done

### API Endpoints #8118
- **CRUD - Connection**: #8127 - Done
- **CRUD - DAG Model**: #8128 - Blocked
- **CRUD - DAG Runs**: #8129 - Development in progress
- **CRUD - Import errors**: #8130 - Done
- **CRUD - Pools**: #8131 - Done
- **CRUD - Task Instance**: #8132 - Blocked
- **CRUD - Variable**: #8133 - Done
- **CRUD - XCOM**: #8134 - Development in progress
- **Log**: #8135 - Done
- **Config**: #8136 - Done
- **Dags structure/Task**: #8138 - Done
- **Extra Links**: #8140 - Done

### HATEOS for API #8117
- Next up

### CRUD Framework for API #8116
- Next up

### Authorization and Permissions #8112
- Next up

### Authentication for API #8111
- Next up

### Custom WEB UI screen to control permissions #8124
- Blocked

### Docs for REST API #8143
- Research in progress

### API security tests #8113
- Blocked
Dev/CI environment
CI environment

- Moved to GitHub Actions
  - Kubernetes Tests ✓
  - Easier way to test Kubernetes Tests locally ✓
- Quarantined tests
  - Fixing the Quarantined tests ✓
- Thinning CI image
  - Moved integrations out of the image ✓
- Future: Automated System Tests (AIP-21)
Dev environment

- Breeze
  - unit testing
  - package building
  - release preparation
  - kubernetes tests
  - refreshed videos

- Code Spaces / VSCode
Backport Packages

- Bring Airflow 2.0 providers to 1.10.*
- Packages per-provider
- 58 packages (!)
- Python 3.6+ only(!)
- Automatically tested on CI
- Future
  - Automated System Tests (AIP-4)
  - Split Airflow (AIP-8)?

Talk: Migration to Airflow backport providers, Anita Fronczak
Thursday July 16th, 4 am UTC
Support for Production Deployments
Beta quality image is nearly ready ✔

Started with “bare image” ✔

Listened to use cases from users ✔

Integration with Helm Chart ✔

Implemented feedback ✔

Dockers Compose

*Talk, Production Docker image for Apache Airflow*

Jarek Potiuk, Tuesday July 14th, 5 am UTC
What’s new in Airflow + Kubernetes
KEDA Autoscaling
KubernetesExecutor
KubernetesExecutor vs. CeleryExecutor

**KubernetesExecutor**
- Dynamic Allocation
- executor_config

**CeleryExecutor**
- Immediate SLAs
- Multiple tasks per-worker
KEDA Autoscaling

- Kubernetes Event-driven Autoscaler
- Scales based on # of RUNNING and QUEUED tasks in PostgreSQL backend
KEDA Autoscaling

\[ \text{CEIL)((0 \text{ RUNNING} + 0 \text{ QUEUED})/16) = 0 \text{ workers} \]
KEDA Autoscaling

CEIL((0 RUNNING + 1 QUEUED)/16) = 1 workers
KEDA Autoscaling

CEIL((20 RUNNING + 20 QUEUED)/16) = 4 workers
KEDA Queues

- Historically Queues were expensive and hard to allocate
- With KEDA, queues are free! (can have 100 queues)
- KEDA works with k8s deployments so any customization you can make in a k8s pod, you can make in a k8s queue (worker size, GPU, secrets, etc.)
KubernetesExecutor
Pod Templating from
YAML/JSON
In the K8sExecutor currently, users can modify certain parts of the pod, but many features of the k8s API are abstracted away.

We did this because at the time the airflow community was not well acquainted with the k8s API.

We want to enable users to modify their worker pods to better match their use-cases.
KubernetesExecutor Pod Templating

- Users can now set the `pod_template_file` config in their `airflow.cfg`
- Given a path, the KubernetesExecutor will now parse the yaml file when launching a worker pod
- Huge thank you to @davlum for this feature
Official Airflow Helm Chart
Helm Chart

- Donated by astronomer.io.
- This is the official helm chart that we have used both in our enterprise and in our cloud offerings (thousands of deployments of varying sizes)
- Helm 3 compliant
- Users can turn on KEDA autoscaling through helm variables
- “helm install apache/airflow”
Helm Chart

- Chart will cut new releases with each airflow release
- Will be tested on official docker image
- Significantly simplifies airflow onboarding process for Kubernetes users
Functional DAGs
Functional DAGs

```python
def get_cat_pictures(num: int) -> List[Dict]:
    response = requests.get("https://cat_pictures.com", params={'num': num})
    return response.json()['cats']

def save_cats(list_of_cats: List[Dict]) -> None:
    for cat in list_of_cats:
        save_it_somewhere(cat)

with DAG("cat_fetcher"):
    get_task = PythonOperator(  
        task_id="get_task", python_callable=get_cat_pictures, op_args=[42]
    )
    cats = "{{ task_instance.xcom_pull('get_task') }}"
    save_task = PythonOperator(
        task_id="save_task", python_callable=save_cats, op_args=[cats]
    )
    get_task >> save_task
```

→ PythonOperator boilerplate code
→ Define separately:
  ◆ order relation
  ◆ data relation
→ Writing jinja strings by hand

- Define separately:
  ◆ order relation
  ◆ data relation

- Writing jinja strings by hand

- Define separately:
  ◆ order relation
  ◆ data relation

- Writing jinja strings by hand
Functional DAGs

Data and order relationship are same!
And works for all operators
Functional DAGs

AIP-31: Airflow functional DAG definition

➔ Easy way to convert a function to an operator
➔ Simplified way of writing DAGs
➔ Pluggable XCom Storage engine

Data and order relationship are same! And works for all operators

Example: store and retrieve DataFrames on GCS or S3 buckets without boilerplate code

Find out more:
AIP-31: Airflow functional DAG definition
by Gerard Casas Saez
10th of July
Smaller changes
Other changes of note

● Connection IDs now need to be unique (#8608)
  It was often confusing, and there are better ways to do load balancing

● Python 3 only ✔
  Python 2.7 unsupported upstream since Jan 1, 2020

● "RBAC" UI is now the only UI ✔
  Was a config option before, now only option. Charts/data profiling removed due to security risks
Road to Airflow 2.0
When will Airflow 2.0 be available?
Airflow 2.0 – deprecate, but (try) not to remove

- Breaking changes should be avoided where we can – if upgrade is to difficult users will be left behind
- Release "backport providers" to make new code layout available "now":
  ```
  pip install apache-airflow-backport-providers-aws \ apache-airflow-backport-providers-google
  ```
- Before 2.0 we want to make sure we've fixed everything we want to remove or break.
How to upgrade to 2.0 safely

- Install the latest 1.10 release
- Run `airflow upgrade-check` (doesn't exist, yet #8765)
- Fix any warnings
- Upgrade Airflow
Thank you!

Time for Q & A